NISTIR 5238

User's Guide for the Programmer's Hierarchical Interactive Graphics System (PHIGS) C Binding Validation Tests (Version 2)

Kevin Brady
John Cugini

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899

NIST

NISTIR 5238

User's Guide for the Programmer's Hierarchical Interactive Graphics System (PHIGS) C Binding Validation Tests (Version 2)

Kevin Brady
John Cugini

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899

August 1993

U.S. DEPARTMENT OF COMMERCE
Ronald H. Brown, Secretary

TECHNOLOGY ADMINISTRATION
Mary L. Good, Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
Arati Prabhakar, Director

Table of Contents

List of Figures

1. C Language Binding Conformance Tests

1.1. Introduction

This document describes the conformance tests for the C binding of the Programmer's Hierarchical Interactive Graphics System (PHIGS) standard (ISO/IEC 9593-4) [CPHIGS][PHIGS89]. This document is an addendum to the "User's Guide for the PHIGS Validation Tests (Version 2.0)" [CGR90]. It is recommended that the User's guide be reviewed before continuing, as the information in this document builds upon information presented in that document (an ASCII version of the User's Guide document is contained in the sub-directory pvt/DOC in this distribution).

The installation of source code, modification of routines, and the running of procedures are covered in the following paragraphs. The document tries to follow a generic approach (i.e., not binding it to any specific system or architecture); the appendices provide examples and procedure files for several specific systems. In general, the document will describe the issues that are relevant to the installation of this software. However, it is impossible to cover all issues relevant to every specific system or circumstance. If, after performing all steps described in this document, you cannot get the test suite to run on your system, feel free to contact the NIST PHIGS Validation Tests (PVT) project leader. Please send all correspondence, including questions about PHIGS validation and obtaining the PVT, to:

Project Leader, PHIGS Validation Tests
National Institute of Standards and Technology
Computer Systems Laboratory
Bldg. 225, Room A-266
Gaithersburg, MD 20899

phone: (301) 975-3265
e-mail: phigs@speckle.ncsl.nist.gov

Certain trade names and company products are mentioned in the text or identified in an illustration in order to adequately specify the experimental procedure and equipment used. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the products are necessarily the best available for the purpose.

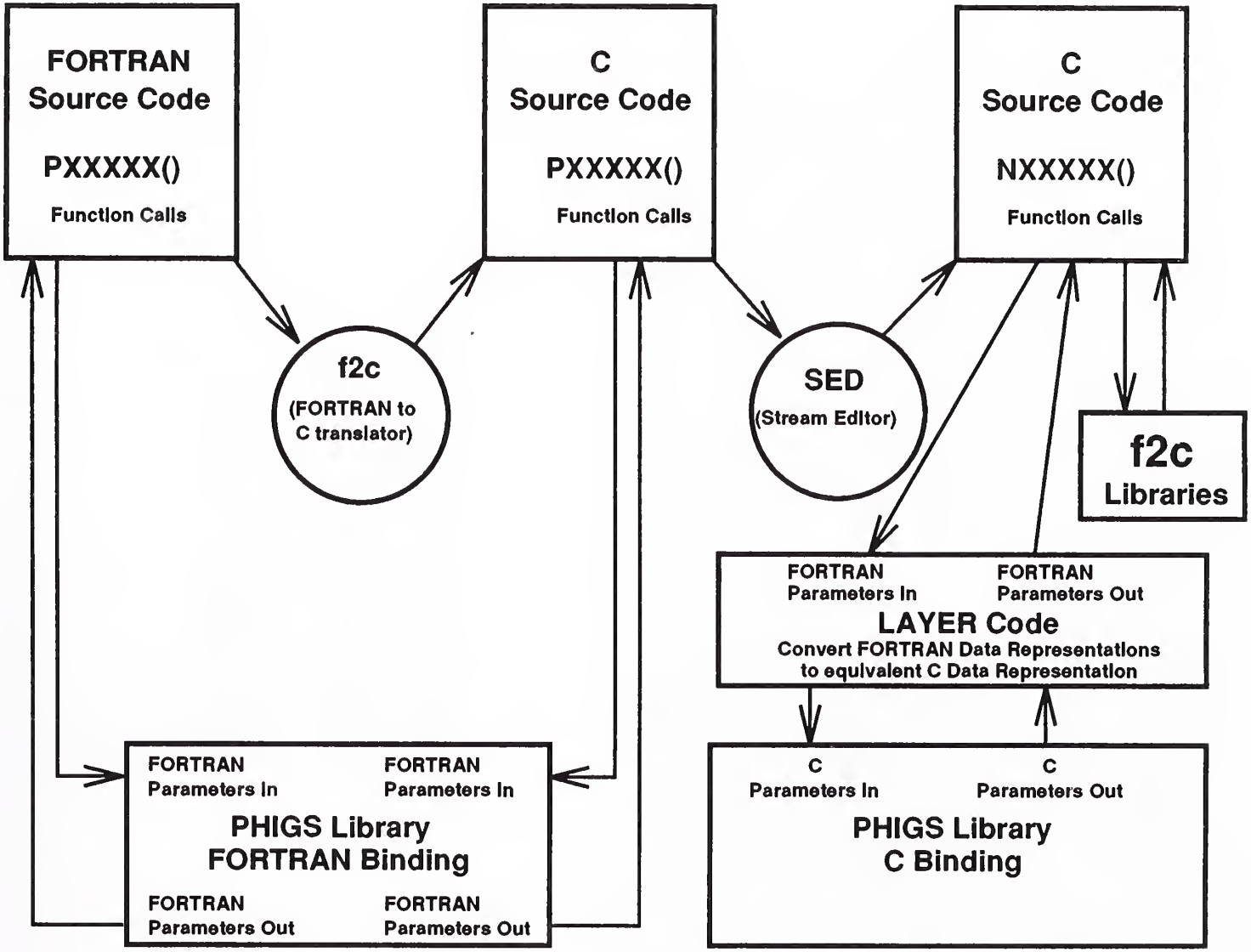

Figure 1.0 PHIGS PVT Source Code Translation Flow Chart

1.2. Software Requirements

This distribution contains the source code for the C PVT test suite, along with the installation and running instructions. The following software is required to run this test suite:

(1) an ANSI C compiler [C1989],

(2) a PHIGS implementation conforming to the PHIGS C binding [CPHIGS],

(3) the f2c (FORTRAN to C) subroutine library (provided in the distribution).

1.3. Translation Utility

The most desirable approach for writing a C binding test suite would have been to start with the design document for each program, and to write new C code to perform the tests outlined. However, due to the limited resources available to the NIST, a FORTRAN to C translator was used to produce the C binding test suite. The translator allowed us to take full advantage of existing FORTRAN test suite code.

Using the translator meant starting with well-designed, well-written, and beta-tested FORTRAN test suite code, and using that code to generate the C binding test suite code. All of the design and programming effort was expended on the FORTRAN code to produce the most efficient and thorough of test suites. The efficiency and thoroughness are carried over to the C binding test suite through the translator with minimal effort.

Once the FORTRAN code was translated by the f2c utility, the generated C code still calls the PHIGS FORTRAN library. To interface this translated code to the PHIGS C library, a "layer" of C code was written between the translated test code and the PHIGS C library (see figure 1.0). Each PHIGS FORTRAN routine has an equivalent layer routine where the first letter has been changed to an "n" (e.g., the PHIGS routine ppl becomes the LAYER routine npl, and is contained in the file npl.c within the LAYER sub-directory). The layer routine accepts the FORTRAN input parameters and converts them into the equivalent C structures required by the C binding. The equivalent PHIGS C library call is then made (e.g., ppolyline for ppl). The returned output parameters from the PHIGS C library call are then extracted and returned in the FORTRAN output parameters. This approach was possible since the standard requires the same information to be present for each function, regardless of the binding. Therefore, even though the syntax of the two calls differ, they both contain the same information.

The translation utility used was the public domain f2c (FORTRAN to C) converter, developed by AT&T and Bellcore labs. After evaluating several proprietary and other public domain packages, this converter was chosen for two reasons:

(1) the C source code generated from the FORTRAN code written thus far required no further "hand modifications",

(2) the purchase of additional software to perform a validation is not necessary.

The directory structure of the f2c utility was altered slightly to decrease the number of libraries created. The libraries libF77 and libI77 have been combined into a single library: libf2c. Since all of the code has been translated, the translation utility f2c itself is not contained in the distribution. The directory pvt/F2C contains the source code for the subroutine library (See Figure 2.0). Since the code has already been translated, *only* the subroutine library (e.g., libf2c) needs to be present to link the programs.

The README file contained in this directory describes the items that need to be checked on a system prior to building the subroutine libraries. Follow the directions in appendix 4.1 on installation.

NOTE 1: The f2c source code itself has not been altered in any way, and sites that have f2c installed as a system library may use their own version.

1.4. Source Code

1.4.1. General

The distribution of source code for the C binding mirrors that of the FORTRAN distribution. A sub-directory under the PVT root called pvt/C contains a /std sub-directory and additional sub-directories below. The tree structure is the same as that of the FORTRAN test suite with the doc.txt files in each sub-directory. The test files (e.g., pxx.c) were created by translating the equivalent FORTRAN files (e.g., the C source file pvt/C/std/04/03/02/p01.c was created from the FORTRAN source file pvt/F/std/04/03/02/p01.f).

1.4.2. Layer Code

In order to use the translated code correctly (i.e., the translated code still calls the PHIGS FORTRAN library), a set of "layer" code routines was developed. Please refer to figure 1.0 for the following discussion.

To translate the FORTRAN code into equivalent C code, it is passed through SED (stream editor) and the names of all PHIGS routines are changed from pxxxxx() to nxxxxx() (e.g., ptx3() becomes ntx3()). The routine nxxxxx() converts the input FORTRAN parameters to the data structures required for the equivalent function call in the C binding (e.g., the routine nrst() converts the input parameters to the data structures required by the C call predraw_all_structs()), allocating all space necessary. The equivalent C call is made to acquire/set the information requested. The routine then extracts the returned data from the C structures and frees any space it previously

allocated. The extracted data is then placed into the FORTRAN output parameters and returned to the program for verification. This small amount of intervention reduces the development time required for the test suite. This code is documented with both the FORTRAN and the C parameters and their meanings. Each of these routines is compiled separately and the resulting object files are assembled into a single library. Since ANSI compliant C allows the use of prototyping [C1989], the layer code provides for both non-prototyped and prototyped source code. The use of a *#define* directive in each source file regulates which function declaration will be used. If your complier supports prototyping, no *#define* directive need be used. If your complier does not support prototyping, all code must be compiled with the directive NO_PROTO. See the installation instructions for further information.

2. Configuration

2.1. Subroutine Libraries

All subroutines written specifically for the PVT test suite have been grouped for distribution by two methods. The hierarchical approach is the method used by the FORTRAN PVT test suite, and maintains backward compatibility for command procedures users may have written. The single directory method combines all routines into a single directory so that a single library may be created. Please use one of these methods, but not both.

2.1.1. Hierarchical Approach

The source code is distributed in a hierarchical structure. The subroutine libraries written specifically for the PVT code reside in certain sub-directories based on usage. Each subroutine library is named sublib.c and resides at the top of the sub-tree of all programs that require it (e.g., if programs 04/03/02/p01, 04/03/03/p01, 04/03/01/p01 each require a common subroutine, that subroutine resides in the file sublib.c located in the sub-directory pvt/C/std/04/03). Each of these sublibs must be compiled and linked for each program that resides below it (i.e., each program must link all sublibs that are encountered as you traverse the tree to the root).

On most systems, you will need to compile the subroutine files (trans_sublib.c and sublib.c in the pvt/C/std and all the local sublib.c files) so as to create subroutine libraries which may then be linked with each test program. Not all test programs use all subroutines. Therefore, you will need to determine what your system requires in order to include necessary subroutines only.

The local subroutine libraries are assigned to nodes of the tree for the purpose of clarifying the logical relationship among the test programs and subroutines. If linking several libraries is difficult in your system, concatenate all subroutine source code as one large file and compile it as one library, preferably in the root. All PVT subroutine

names are unique, thereby avoiding name replication problems.

2.1.2. Single Directory Approach

Each of the subroutine libraries is also distributed in a single directory under the PVT root, pvt/V2LIB (See Figure 2.0). This allows easier generation of object libraries. The sub-directory pvt/V2LIB contains all routines, one routine per file, each file name being the routine name. Files ending in .c are the C versions translated form the equivalent FORTRAN versions (e.g., XPOPPH.c was translated from XPOPPH.f). Each of the files must be compiled and assembled into a library for linking. All object files resulting from the compilation of source files ending in .c should comprise a single library. It is not recommended that the files be separated into multiple libraries as this will complicate linking because the calling order of the routines must be followed.

2.2. Customizations

Some of the source code will need to be changed to run on your system. The first change, naming the PVT configuration file, is mandatory for all systems. All other changes are optional. The changes are explained in this section for convenience only; see the installation procedures in appendix 4.1 which cover this topic in detail.

(1) Naming the PVT Configuration File

You must choose a name for the PVT configuration file. The name you select must be absolute (i.e., it must be valid when used from any part of the hierarchy). It is recommended to locate the PVT configuration file in the PVT root.

This name must be inserted in three locations:

(a) the initph.c program which writes the configuration file (located in the pvt/C/std sub-directory),

(b) the INITGL routine which reads the configuration file and is located either in the global pvt/C/std/sublib.c file, or in pvt/V2LIB/INITGL.c,

(c) the MULTWS routine which also reads the file, located either in the global pvt/C/std/sublib.c file, or in pvt/V2LIB/MULTWS.c.

In all three cases, search for the string "INITPH$DAT" (the default name) in these routines to locate the insertion point.

The PVT configuration report file is a human-readable version of the PVT configuration file. You must also choose a name for this file as well, such as "initph.prt", and insert it into the initph.c program.

(2) Special Processing for Opening PHIGS

All test cases call XPOPPH to open PHIGS, rather than the standard POPPH.

The XPOPPH subroutine, as delivered, simply calls POPPH. If your system has special processing requirements for accessing PHIGS, these may be addressed by additions to XPOPPH.

NOTE 2: For validation purposes, all changes are subject to approval by NIST.

(3) Resolution of Parameters for <Open Workstation>

The INITGL and MULTWS subroutines read the PVT configuration file in order to determine the parameter values needed to open the primary and secondary workstations (connection identifier, and workstation type), and report these back to the calling program. INITGL sets the values of variables globally to do this, while MULTWS returns the values in its output parameters. In both cases, the assumption is that the correct values are static and can be set once by the INITPH procedure (see below). If your system is such that this information can be determined only at run-time, you must re-code the relevant sections of XPOPPH and MULTWS so that they still deliver the required values.

XPOPPH This routine is called to open PHIGS for the PVT tests. It is provided for any implementation that must perform special tasks prior to the opening of the PHIGS workstation (e.g., derive workstation type or connection identifier). Although the call to open the PHIGS workstation itself is standardized, the workstation type and connection identifier (two of the parameters) are implementation dependent. If your implementation represents these parameters as static integers known prior to run-time, no modifications are necessary. This information will be stored in the initialization file by the program INITPH. However, if your implementation represents these as non-static integers, (known only at run-time), modifications are necessary to allow the workstation type and connection identifier to be computed correctly. This requires the correct code to be added to generate the correct workstation type and connection identifier for the "open workstation" call.

(a) Assign to the variable *globnu_1.wtype* the correct integer value that represents the workstation type.

globnu_1.wtype = <your workstation type>

(b) Assign to the variable *globnu_1.conid* the correct integer value that represents the connection identifier (see source for XPOPPH.c).

globnu_1.conid = <your connection identifier>

These assignments are critical, because this places the workstation type and connection identifier in global variables that will be used in subsequent calls to "open workstation".

MULTWS This routine is called to retrieve information about multiple PHIGS workstations for the PVT tests. Again, the workstation type and connection identifier (two of the parameters) are implementation dependent. If your implementation represents these parameters as static integers known prior to run-time, no modifications are necessary. This information will be stored in the initialization file by the program INITPH. However, if your implementation represents these as non-static integers (known only at run-time), modifications are necessary to allow the workstation type and connection identifier to be computed correctly. This requires the correct code to be added to generate the correct workstation type and connection identifier for the "open workstation" call. At the end of the file MULTWS.c, before the assignments to the ouptut variables *owkid*, *oconid*, *owtype* and *owcat*, add code segments that will perform the following:

(a) Assign to each element of the array *mwtype* the correct integer value that represents the workstation type, up to the number of workstations you are using.
mwtype[i] = <your workstation type>, i = 0..number of workstations

(b) Assign to each element of the array *mconid* the correct integer value that represents the connection identifier, one for each connection identifier.
mconid[i] = <your connection identifier>, i = 0..number of workstations

These assignments are critical, because this places the workstation type and connection identifier in output variables that will be used in subsequent calls to "open workstation".

(4) Naming Individual Message File

If you request that the test programs generate individual message files (see section 4.2.3 of the User's Guide) INITGL will, by default, form the name of the file by using "p" as a prefix, the two-digit ordinal number of the program, and ".msg" as the extension; e.g., p04.c will write to p04.msg. If you prefer another naming convention, search for ".msg" in the INITGL routine and change the code accordingly.

(5) Random Number Generator

Since the C language provides a standard interface to the time function [C1989],

a variable seed was available for the random number generator. No modifications should be necessary to the file RND01.c. All the other random-number routines are built on RND01, and should not need to be changed.

(6) Providing Valid Names for Archive Files

The routine AVARNM in node 03 of the PVT tree must return to the caller an integer representing the valid name of an available empty archive file. The code assumes that this name is a FORTRAN logical unit number. If your system has a different interpretation, or has special requirements for opening an archive file, you must modify this subroutine accordingly. The translator will convert FORTRAN logical unit numbers into a character string representing the filename (see section 3.3).

(7) Time-stamping Message Files

The original code developed for the tests is written in FORTRAN, and the FORTRAN standard provides no function for determining time or date [FORT78]. The C language does provide a standard method for determining the date [C1989]. If you wish to include this information in the PVT output, alter the INITGL subroutine at the point where it formulates the header system message. This is done at the end of the subroutine in the last call to BRDMSG.

(8) Operator Communication

The OPMSG and OPYN routines write messages to and read messages from the operator. Because a workstation may not be open at the time these are executed, the PVT code resorts to the use of FORTRAN's print and read statements which are converted into the equivalent C routines (i.e., printf/scanf). If there is a better way to send a character string to and from the operator in your system, you may re-code these routines accordingly. They are located either in the pvt/C/std/sublib.c file or in the pvt/V2LIB sub-directory. No change is necessary if print and read work well within your system.

(9) End of File

If you specify a global message file (see section 4.2.3 of the User's Guide) the INITGL routine in sublib.c must position the file pointer at end of file so as to append new messages. In standard FORTRAN [FORT78], the entire file must be read to position the file pointer correctly. Since the translator uses the FORTRAN code, this inefficiency is carried into the generated C code. If you wish to provide a more efficient way in C, you may substitute the method for the one provided in INITGL. Otherwise, no change is needed.

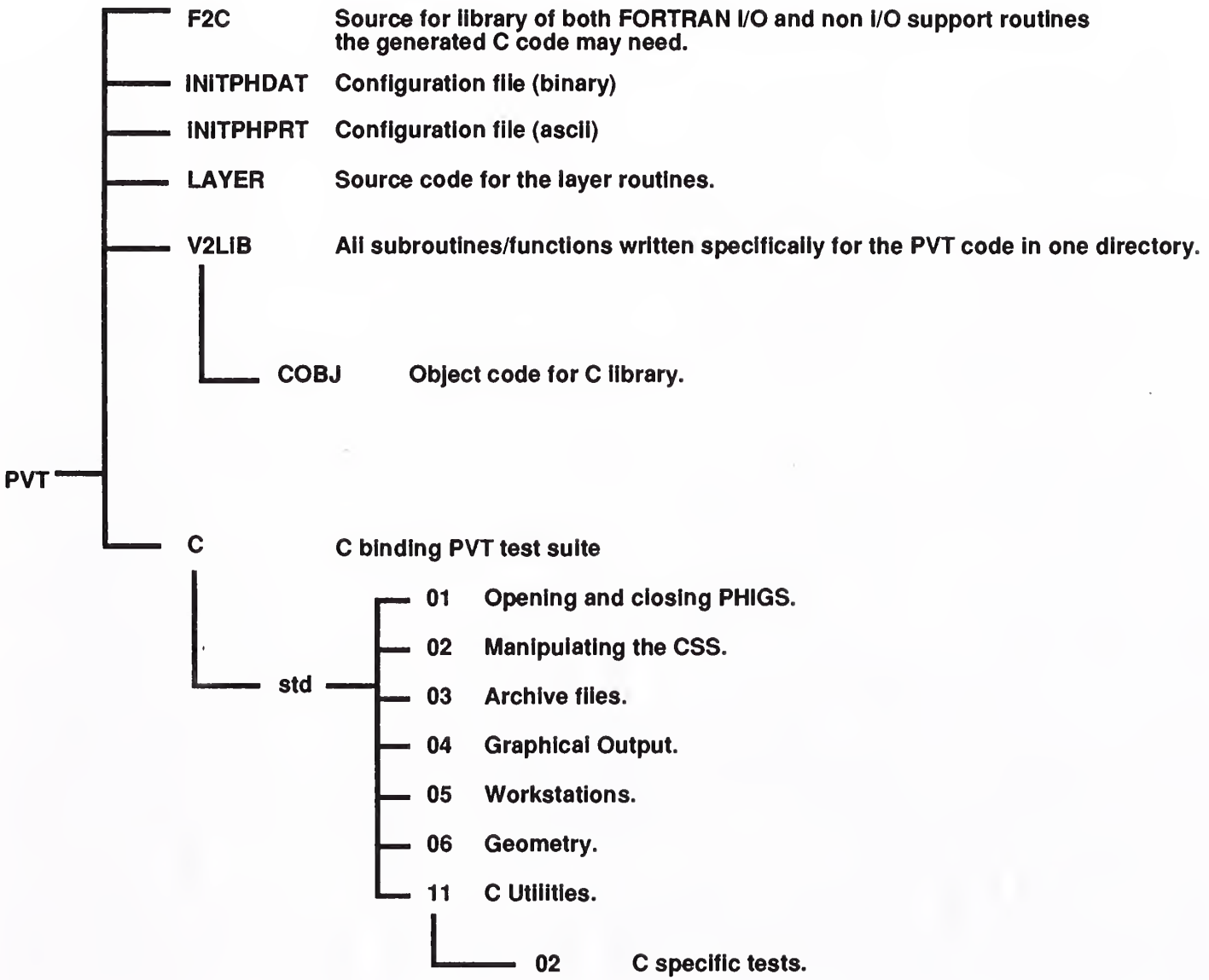

Figure 2.0 File Structure Hierarchy

2.3. Linking Test Programs

It is strongly recommended that you set up a command procedure to compile, link, and execute a test program solely by referring to the name of the program. In particular, all local subroutine libraries (those above the program in the directory tree) and the global libraries must be made available, as well as linking to the code of the PHIGS implementation.

The following libraries are required for the test suite:

- the PVT library, (e.g., the sublibs or the files in V2LIB)
- the layer code library, (e.g., all the nxxxxx routines)
- the f2c library, (e.g., libf2c)
- the math library, (e.g., the standard library)
- the PHIGS library, (e.g., your PHIGS library to be validated)

The following libraries may be required for the test suite depending on the implementation:

- any additional library required by your implementation. (e.g., X11 libraries)

The following order is important for those linkage editors that do not have wrap-around search when resolving external references:

<test file> <PVT library> <layer code> <libf2c> <math> <your PHIGS libraries>

2.4. Running Test Programs

Each of the programs in the sub-tree must be executed to perform the validation. Any procedure that can be developed on your system to accomplish this is recommended. When the actual validation is performed, each program must compile, link and execute without error.

The program INITPH in the pvt/C/std sub-directory must be run prior to the execution of any test (pxx). Follow the installation procedures contained in appendix 4.1 for instructions on the information required by this program. INITPH will prompt the operator for a few site specific terms to be used by the test suite (e.g., the workstation type). Once this program has completed, the test suite may be executed. The order of the tests is insignificant; because they are independent. The only requirement is that they all pass. A common grouping of passive tests (requiring no operator intervention) and active tests (requiring operator intervention) may be desired. See the file pvt/C/std/pgm_char.prt for a list of which programs are active tests and which are passive tests.

The sub-directory pvt/C/std/11 (see figure 2.0) contains programs that are relevant only to the PHIGS C binding. The 02 sub-directory contains 5 programs that test

portions of the C binding. Program p01 tests for the existence and correct value of each *#define* directive required by the standard. This program must be compiled so that it includes the phigs.h file provided with your implementation. Programs p02, p03, and p04 test for the correct typedef and external definitions required by the standard. See the operator script in the 11/02/doc.txt file for details. Finally, program p05 tests the create_store and delete_store routines.

2.5. Debugging

As previously stated, the C code is the translated version of the equivalent FORTRAN code. The converter translates the FORTRAN source code, but the readability of the code is lost in the translation. If the tests run successfully, the user should never have to look at the translated C code to see what the code is actually doing. However, if a failure or an abort does occur and it is necessary to see exactly where the failure occurred, perform the following:

(1) Look at the program design document contained in the file "doc.txt" in the same sub-directory as the program that failed. This document describes the part of the standard that is being tested and the methodology the test is using.

(2) Look at the source code for the layer routine that failed. This code resides in the sub-directory pvt/LAYER (See Figure 2.0). Each PHIGS FORTRAN routine has an equivalent layer routine where the first letter has been changed to an "n" (e.g., the PHIGS routine ppl becomes the LAYER routine npl, and is contained in the file npl.c). The input parameters along with the output parameters are explained for both the FORTRAN and C languages. Print statements can be inserted easily into this code for debugging/verifying failures.

(3) Look at the translated source code (pxx.c). This code can be confusing, even to an experienced C programmer, and reviewing the code is recommended only as a last resort. The layer code and the design document should provide ample means of debugging/verifying failures.

3. Helpful Information

This section describes a few of the problems encountered in the design and implementation of the test suite. The design philosophies inherent to the translator directed the development of the layer code.

3.1. File Naming Conventions

In FORTRAN, a file can be opened prior to its usage by a PHIGS subroutine call. The PHIGS call checks if the file has been opened, and if not, opens it. This method

does not translate well into the C language. However, the method described in 3.3 allows the C code to emulate the FORTRAN code.

The file, as an input parameter, appears in <open-phigs>, <open-archive-file>, <error-handler>, <error-logging>. The file, as an output parameter, appears in <inquire-open-archive-file>. The PHIGS FORTRAN binding assigns the filename to the type integer, and in most instances is associated with a logical unit number. The PHIGS C binding assigns the filename to a character string. This difference was solved by the method described in 3.3.

3.2. Prototyping

Prototyping is allowed by the C language standard and all code generated will, by default, use it [C1989]. However, for those compilers that do not support prototyping, the following steps will allow non-standard code to be generated.

(1) Use the directive *#define* NO_PROTO when compiling all code, by changing the "Makefiles" in the PVT root, in V2LIB, and in the LAYER directories.

(2) Run the tests as described above.

3.3. File Handling

FORTRAN addresses files using the integer unit number with which the file was opened. The C language, on the other hand, addresses a file by the file pointer with which it was opened. These two types differ greatly and there is no one-to-one correspondence between them. The C tests must use a table that is kept by the f2c code which keeps track of FORTRAN unit numbers and the C files they represent. A number of the layer code routines (i.e., nxxxxx()) include the f2c header file "fio.h". These routines deal with file handling and must search the file table to match a FORTRAN unit number to a C filename. An array (of default size 100) is created at the start of each program. Array index [i] contains the information on logical unit i (e.g., array[10] contains the information on logical unit 10). This information should not affect any routine provided in the test suite, for they correctly handle each situation. This information is provided for debugging and informational purposes only.

3.4. Special Characters

The test suite makes use of the entire printable ASCII character set (i.e., one of the requirements of the standard). The characters appear in the source code and may also have another meaning on some systems (e.g., the backslash "\" character in UNIX). Most compilers have a switch/parameter that notifies the compiler to ignore the special "system" meaning of these characters during compilation. If the code does not compile, check if your system uses any special characters, and enable the switch/parameter for your compiler that ignores the special meaning.

3.5. Pack/Unpack

The FORTRAN binding has two functions, *pack* and *unpack*, which convert data from arrays of integer, real and character data to and from an array of 80 character records. This latter array, in turn, may be used as a parameter to subsequent PHIGS calls. However, the C binding has no such functions (it uses large structures to accomplish this same function). Therefore, the layer code for *pack* and *unpack* does not encode the data into the character arrays. Rather, it defines a general purpose data type (see struct.h in pvt/LAYER) to hold such data.

The layer code for *pack* and *unpack* (*nprec* and *nurec*) will move data into and out of the input arrays to a structure defined in struct.h, contained in the LAYER directory. Since the NIST is coding *pack* and *unpack* (not the implementor), the format for data records is set by us. The current approach is to overlay a structure (special typedef) which holds 20 integers, 20 reals, and 5 strings on top of the raw 80xN area defined in FORTRAN. This method requires that 592 bytes of storage space be available, resulting in the minimum datrec declaration being 80 x 8. Since the test code adheres to this requirement and the method simulates the FORTRAN implementation exactly, the data storage will be totally transparent to the user.

The layer code for functions having input parameters of type *data-record* must be aware of the datrec format and use it correctly. These functions then face only the familiar problem of re-formatting FORTRAN style data into C style data to be passed to the equivalent C functions. The only difference is that in this case, *NIST* defined the FORTRAN-style data, instead of the standard. This makes sense, since the FORTRAN binding specifically does *not* define the internal format of data-record - it mandates only that *pack* stores the data, and that *unpack* is capable of retrieving the same data.

Using the C language, this new data structure is defined as a parameter to each of the functions that use it, instead of declaring it as the FORTRAN 80xN array of characters. Since the FORTRAN main program does not manipulate these arrays in any way (it only passes them to the subroutines), the actual content of the array is never known by the main program. The subroutines, however, receive the array as a pointer to a structure (defined in struct.h). The data from the arrays is stored in the structure and the array passed back. This array is passed by the main program to one of the PHIGS functions that uses it. Each of those functions again declares the received array as type pointer to structure (e.g. a structure pointer), and interprets the structure the same way it was packed, allowing the data to be retrieved and used. There is overhead in this method since FORTRAN requires two steps (pack data, use data), and C requires only one (use data). The C code must therefore emulate the FORTRAN code to limit the amount of changes that must be performed by hand on translated code.

3.6. Strings

Strings are represented differently in the FORTRAN and C languages. In FORTRAN, the length of a character variable is part of the variable itself. This length information is hidden from the user and required only by the machine. In C, a string can be of any length and a null termination character signals the end of the string. The translator, therefore, represents each FORTRAN character variable as two C data types. The first is the length of the string, and the second is the string itself. Each subroutine call in FORTRAN that contains a character variable is translated into a C function call with an extra parameter for each character variable added on to the end. The lengths of each string are conveyed in these extra variables. The layer code is written accounting for these length variables. In the LAYER code, some PHIGS functions have more parameters than the standard requires. These parameters are added to hold the lengths of the strings to be received. For example the FORTRAN call to write text:

```
CALL PTX ( PX, PY, CHARS )
  REAL        PX, PY
  CHARACTER*(*) CHARS
```

Is translated to:

```
ntx (*px, *py, *chars, clen)
  float *px, *py;
  char *chars;
  int clen;
```

Notice the extra parameter *clen*. This variable is added to convey the length of the character string *chars*. Each routine that uses strings follows this convention of the translator. Since the length is known, the layer code checks for variables that are too small to hold data returned from inquires, and will return the PHIGS error 2001 (Ignoring function, output parameter size insufficient) if they are found.

3.7. Error Handling

Error handling is also treated differently in the two languages. The FORTRAN standard defines the subroutine *PERHND* as the name of the error handling routine. A user can write his own error handling subroutine, but must call it *PERHND*. To have this routine invoked, the user links his *PERHND* routine before the system routines. The C standard defines the function perr_hand to perform the equivalent error handling function. However, the routine name can be changed by the function *pset_err_hand*. *Pset_err_hand* takes as one of its parameters the name of the function to be called instead of the perr_hand function. The FORTRAN test code makes extensive use of the *PERHND* routine we had written, determining if a failure is so great that the test

code must abort immediately. In order to reuse this routine in the translated code, the routine *PERHND* could have been changed to perr_hand. However, the parameter types differ between the two languages (i.e., error file) and would have required modification by hand after it had been translated. The solution chosen inserts a call to *pset_err_hand* and changes the name of the routine called to be *nerhnd* (contained in the LAYER directory). The routine *nerhnd* performs the opposite of what all the other layer routine calls do, it translates the C parameters to their equivalent FORTRAN parameters; and calls the *PERHND* function we had written. Again, this change should be totally transparent to the user.

Normally, re-naming and re-parameterization are affected only by the layer code. This instance is unique in that control of the calling of the routine *PERHND* does not lie in the test suite code, but rather in the PHIGS/C functions. Some linkage editors will not load the object code for *PERHND* at link time since it is not explicitly called by the test code, but is called internally by the PHIGS code. To circumvent this, we have explicitly added a call to *PERHND* to the routine INITGL to force the object code to be loaded. To ensure that *PERHND* is not called from INITGL, we have encapsulated it with an *if* condition that will never be executed. Be advised however; some compilers will remove this call if optimization is in force.

3.8. Parameter Passing

Another common problem encountered when changing between languages is the way parameters are passed to functions. FORTRAN parameters are passed by reference or value, depending on whether the parameter is a variable or expression. The FORTRAN standard states that constants (e.g., 3), expressions (e.g., X+2) and constant expressions (i.e., variables declared as parameters), declared as parameters to functions, may not be changed in those functions. This would require parameters to be passed by value. However, it is not specified that they cannot be passed by reference. To avoid this problem, the translator passes **all** variables by reference. To prevent problems with constants and expressions, they are first assigned to a newly created temporary variable, and then the address of that variable is passed to the function. As an example:

```
CALL SUB (3, X, X+3)      { value, reference, value }
```

Is translated to:

```
int c_1, c_2;
c_1 = 3;
c_2 = X + 3;

sub(&c_1, &x, &c_3);
```

Arrays and regular variables are already passed by reference, and there is no distinction between input or output variables. The layer code receives **all** variables as pointers.

3.9. Array Indexing

The last problem encountered dealt with array indexing which is the result of another difference between the C and FORTRAN languages (i.e., because of the way they physically store arrays). The FORTRAN standard specifies column-major order when physically storing arrays in memory. C, on the other hand, specifies row-major order when storing the arrays. If a FORTRAN program passs an array to a C function, the array has to be transposed first. Then, the array can be used by the C routines, and again transposed before being passed back to the FORTRAN program. The translator emulates the physical storage of the FORTRAN code by collapsing all n-dimensional arrays to a single dimension, and calculating array indexes internally. The layer code performs all necessary transpositions when building the equivalent C structure. The layer code, however, does receive some arrays as two-dimensional, but does the array indexing correctly.

4. Appendix

4.1. UNIX Systems

The following steps must be followed in order to install the test suite on a UNIX system. Note that Makefiles have been provided that will run on most systems but may require modifications for your system.

NOTE 1: No executable is shipped in the distribution; all programs must be regenerated. When building the f2c libraries, please read the section describing the use of *#define* USE_STRLEN. This definition is crucial for some systems.

Step [1] Build the f2c library - libf2c.a

(a) cd pvt/F2C

(b) Edit the makefile

- Check the description of onexit and how it applies to your system.
- Select the C compile line that you want (i.e., whether you want prototyping).

(c) Review the README file and modify items as required by your system.

(d) Type *make*

NOTE 2: This is a condensed version of the actual f2c utility. The libraries libF77 and libI77 have been combined into one library, and the actual translator f2c is not distributed. All code has been translated and only the subroutine library is required. The code itself has not been altered in any way, and sites that have f2c installed as a system library may use their own version.

NOTE 3: This has been successfully built on a SPARC station, VAX VMS ALPHA, and PC DOS. Makefile.vms is a command procedure for VMS systems.

Step [2] Build the C layer code library - layer.a

(a) cd pvt/LAYER

(b) Edit the Makefile

- Change PVTHOME (currently /home/kevin/pvt) to the directory where the PVT test suite is installed.
- Change XINCDIR (currently /usr/openwin/include) to the directory where X11 is installed. This is required for include files if your implementation uses X11. If you do not require this variable, just leave it blank.
- Change PHIGSINCDIR (currently $(PHIGSHOME)/include/phigs) to the directory where PHIGS is installed. This is required for include files.
- Select the C compile line that pertains to your system (i.e., whether you want prototyping)

(c) Type *make*

Step [3] Build the subroutine library - libcpvt.a

(a) cd pvt/V2LIB

(b) Edit the Makefile

- Change PVTHOME (currently /home/kevin/pvt) to the directory where the PVT test suite is installed.
- Change XINCDIR (currently /usr/openwin/include) to the directory where X11 is installed. This is required for include files if your implementation uses X11. If you do not require this variable, just leave it blank.
- Change PHIGSINCDIR (currently $(PHIGSHOME)/include/phigs) to the directory where PHIGS is installed. This is required for include files.
- Select the C compile line that pertains to your system (i.e., whether you want prototyping)

(c) Edit INITGL.c:

- Change the filename character variable on line 180 to the directory where you will install the configuration file using initph.
  Currently:
  s_copy(filenm, "INITPH$DAT", 60L, 10L);
  Change to:
  s_copy(filenm, "<your directory>/pvt/INITPH$DAT", 60L, <NN>L);
  Change the <NN> to the length of the character string, (up to a max of 60)

(d) Edit MULTWS.c:

- Change the filename character variable on line 206 to the directory where you will install the configuration file using initph.
  Currently:
  s_copy(filenm, "INITPH$DAT", 60L, 10L);
  Change to:
  s_copy(filenm, "<your directory>/pvt/INITPH$DAT", 60L, <NN>L);
  Change the <NN> to the length of the character string, (up to a max of 60)
- Refer to the discussion on MULTWS in the customizations section (section 2.2) to determine if further changes are required (i.e., do you need to generate a special workstation type).

(e) Edit XPOPPH.c:

- Refer to the discussion on XPOPPH in the customizations section (section 2.2) to determine if any changes are required (i.e., do you need to generate a special workstation type).

(f) Type *make*

Step [4] Build initph and oprcmt

(a) cd pvt

(b) Edit the Makefile

- Change PVTHOME (currently /home/kevin/pvt) to the directory where the PVT test suite is installed.
- Change XINCDIR (currently /usr/openwin/include) to the directory where X11 is installed. This is required for include files if your implementation uses X11. If you do not require this variable, just leave it blank.
- Make sure PHIGSHOME is set.
- Check the system variable CPHIGS_LIB. It must be set to the libraries required by your PHIGS implementation, including the PHIGS libraries themselves (i.e., does it use any X libraries?). Set this variable accordingly to locate the libraries you will need.

(c) Edit pvt/C/std/initph.c:

- Change the filename character variable on line 361 to the directory where you would like to have the configuration file installed.
  Currently:
  s_copy(filenm, "INITPH$DAT", 60L, 10L);
  Change to:
  s_copy(filenm, "<your directory>/pvt/INITPH$DAT", 60L, <NN>L);
  Change the <NN> to the length of the character string, (up to a max of 60)

[d] Change the filename character variable on line 363 to the directory where you would like to have the HUMAN-READABLE configuration file installed.
Currently:
s_copy(filenm, "INITPH$PRT", 60L, 10L);
Change to:
s_copy(filenm, "<your directory>/pvt/INITPH$PRT", 60L, <NN>L);
Change the <NN> to the length of the character string, (up to a max of 60)

(e) Type *make* cadmin

Step [5] run pvt/C/std/initph to create the configuration file.

The PVT configuration file contains information which is specific to the PHIGS implementation being tested. This file is used by all test programs. Its purpose is to allow an operator to specify such information only once at the beginning of each session, rather than repeating it for each program. The initph program creates this file using input received from the operator's responses. The program is stored as initph.c in the pvt/C/std directory. It uses some subroutines from the global subroutine library, and also from PHIGS itself, so these libraries must be available as it is compiled and linked. Execute initph, and respond as prompted. Most responses are in the form of an integer. Be prepared to supply the following information to initph:

[a] parameters for <open phigs> (error file and memory units),
The first two questions concern the input parameters to be passed to the <open phigs> function whenever that function is needed in a PVT program. Supply the values your implementation requires.

[b] number of workstations accessible in this session,
Tell initph the total number of accessible workstations (primary and secondary).

[c] <open workstation> parameters for each accessible workstation,
For each of the workstations from the previous question, supply the values your implementation requires for the input parameters to be passed to the <open phigs> function whenever that function is needed in a PVT program (i.e., workstation identifier, connection identifier, and workstation type). Be sure that the first set of parameters refers to the primary workstation.

[d] whether to suppress 'pass' messages,
Initph will ask whether you want a message to be generated whenever the implementation successfully passes a test case (TC) in a test program. You can specify either that such messages are always suppressed, always generated, or that each program asks the operator at run-time whether pass-messages are to be suppressed. No other type of message may be suppressed.

[e] choice of destination(s) for messages (screen, individual files, or global file),

Indicate whether messages are to be sent to the operator (typically on the screen). Next, indicate to which files messages should be written. Individual message files are created once per execution of a test program. By default they are given the same name as the program, but with a "msg" suffix, rather than "f". The global message file is a cumulative file to which messages are appended whenever a test program is run. These are independent choices; messages can be sent to any combination of the three destinations: operator, individual file, or global file. Each enabled destination receives exactly the same set of messages.

[f] logical unit numbers for individual message files (if used),

Since some operating systems have reserved logical unit numbers in FORTRAN, you are asked to provide these for the individual and/or global file, if they have been designated as destinations. The C code will associate logical unit numbers with file names, so the actual integer value chosen will have no meaning. Choose an integer between 1-100.

[g] file name for global message file (if used),

If you specify a global file, you must provide an absolute name for this file, so that all programs can write to it. You may want to specify a distinct name for the global message file of each PVT session. This response is not in the form of an integer, as are the others.

[h] maximum line length for messages,

You must specify the maximum number of characters per line which should be generated when the PVT system formats a message. Some messages may be quite long and would not fit on a reasonably-sized single line. Message text is never truncated; rather, it is simply broken into lines of the specified size.

[i] test with a pseudo-random or true random number sequence,

Many of the interactive tests randomize the choices presented to the operator so that the correct responses are not predictable (see section in User's Guide on operator interaction). For some purposes, however, it is desirable that the tests execute with repeatable displays and operator prompts. If you want to get repeatable behavior, enter a real number between 0.1 and 0.9. This value will be used as the seed for a random number sequence. Thus, re-initializing to a distinct value between 0.1 and 0.9 will cause repeatable behavior within the new session, but distinct from that of the previous session. Entering any value outside the range of 0.1-0.9 causes the system to use a time function to set the seed for the random sequence, and thus generates truly random operator choices.

[j] whether the primary workstation is capable of visual output,

If the primary workstation is capable of visual output, answer yes, otherwise no. The normal answer is "yes". The question is here to allow for testing of INPUT-

only workstations and metafile workstations in later versions of the PVT.

[k] method for prompting the operator, and maximum line length for prompts,

When running the interactive tests, the system poses questions to the operator. This option lets you choose the mechanism for transmitting those questions: 1-FORTRAN print, 2-PHIGS <message>, or 3-PHIGS <text>. The FORTRAN PRINT statement is translated into the C printf statement, and will write to a separate window. PHIGS <text> will use some of the PHIGS display space for questions, possibly leaving less room for the picture under examination. If using FORTRAN print or PHIGS <message>, you must specify the maximum number of characters per line in interactive prompts.

[l] method for operator responses to prompt,

You may specify the means by which the operator is to respond to prompts: either FORTRAN-read or PHIGS <request string> (and the device number for <request string>). The FORTRAN read is translated into the C scanf function, and will normally read at a separate window (in a windowing environment).

[m] location and size of the dialogue and echo areas,

If you are using PHIGS for operator input or output, you must specify where the dialogue area (containing operator prompts and responses) should be. The choices are: 1-dialogue at bottom of screen, 2-dialogue at right, or 3-toggle picture and dialogue so that the screen can alternate between the two. Since the picture area will use the largest square remaining on the screen after the dialogue area has been reserved, it is recommended that the dialogue area be on the right for wide screens, and on the bottom for tall screens. You must then specify what percentage of the screen should be reserved for the dialogue area. Some value in the range of 15-30 percent is usually a reasonable choice. If you are using PHIGS for operator input, specify what percentage of the dialogue area is to be reserved for the echo of operator responses (the remaining area is used for prompts). Since prompts are usually larger than the responses, some value like 10-20 percent is a good choice.

[n] ratio of meters to DC (Device Coordinates) units for the primary workstation,

You must tell the system the ratio of meters to DC units for this workstation. You may either enter the number directly, or physically measure a diagonal line that will be drawn on the screen by initph using PHIGS, and let initph compute the result.

At the successful conclusion of initph, the operator receives a report on the names of the files to which the PVT configuration file and PVT configuration report file (the human-readable version) have been written.

Step [6] Build the PVT test suite.

[a] Edit pvt/Makefile:

- Select the C compile line that pertains to your system (i.e., whether you want prototyping)

In the PVT root type one of:

make all	[Build the C test suite]
make cadmin	[Build the C version of initph]

Step [7] Run the test suite.

Each of the programs that exist in the sub-tree must be executed to perform a complete validation. You should develop procedures on your system to accomplish this efficiently. When the actual validation is performed, each program must compile, link and execute without error.

In the PVT root type one of:

make -f Makefile.run run	[Run both active and passive C tests]
make -f Makefile.run runactive	[Run active C tests]
make -f Makefile.run runpassive	[Run passive C tests]

5. REFERENCES

[CGR90] John Cugini, Mary T. Gunn, Lynne S. Rosenthal, *User's Guide for the PHIGS Validation Tests (Version 2.0)*, NISTIR 4953, National Institute of Standards and Technology, Gaithersburg, MD, 1990.

[PHIGS89] *Computer Graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS) (Part 1: Functional Description)* , ISO/IEC 9592-1:1989, American National Standards Institute, New York, NY, 1989.

[CPHIGS] "*Information processing systems - Computer Graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS) language bindings - Part 4: C* , ISO/IEC 9593-4:1990, American National Standards Institute, New York, NY, 1990.

[C1989] *Programming Language C*, ANSI X3.159-1989, American National Standards Institute, New York, NY, 1989.

[FPHIGS] *Information processing systems - Computer Graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS) language bindings - Part 1: FORTRAN*, ISO/IEC 9593-1:1990, American National Standards Institute, New York, NY, 1990.

[FORT78] *Programming Language FORTRAN*, ANSI X3.9-1978, American National Standards Institute, New York, NY, 1978.